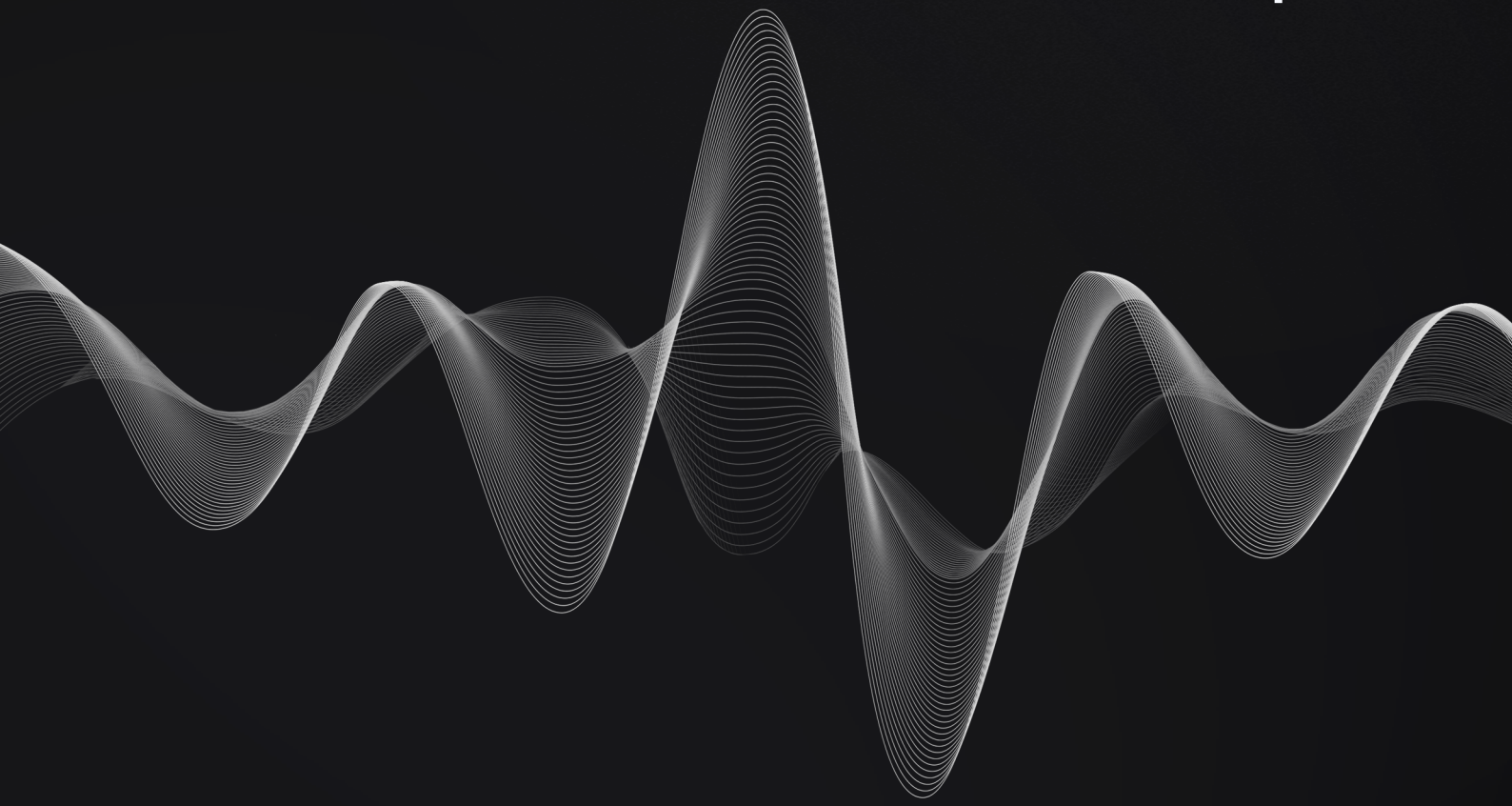
RESONANCE

RTLabs
Demether Protocol Audit Report



Blockchain, Emerging Technology, and Web2
CYBERSECURITY PRODUCT & SERVICE ADVISORY

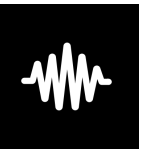
Document Control

PUBLIC **FINAL**(v2.0)

Audit_Report_DMTH-PRO_FINAL_20

Aug 4, 2024	v0.1	João Simões: Initial draft
Aug 5, 2024	v0.2	Michal Bajor: Added findings
Aug 5, 2024	v1.0	Charles Dray: Approved
Aug 26, 2024	v1.1	João Simões: Reviewed findings
Aug 31, 2024	v2.0	Charles Dray: Published

Points of Contact	Kartik Swaminathan	RTLabs	k@rtlabs.xyz
	Jack Shi	RTLabs	jack@rtlabs.xyz
	Charles Dray	Resonance	charles@resonance.security
Testing Team	João Simões	Resonance	joao.simoes@resonance.security
	Ilan Abitbol	Resonance	ilan.abitbol@resonance.security
	Michał Bazyli	Resonance	michal.bazyli@resonance.security
	Michal Bajor	Resonance	michal.bajor@resonance.security



Copyright and Disclaimer

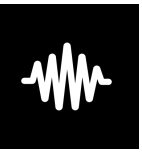
Contents

Executive Summary

RTLabs contracted the services of Resonance to conduct a comprehensive security audit of their smart contracts between July 26, 2024 and August 5, 2024. The primary objective of the assessment was to identify any potential security vulnerabilities and ensure the correct functioning of smart contract operations.

During the engagement, Resonance allocated 2 engineers to perform the security review. The engineers, including an accomplished professional with extensive proficiency in blockchain and smart-contract security, encompassing specialized skills in advanced penetration testing, and in-depth knowledge of multiple blockchain protocols, devoted 7 days to the project. The project's test targets, overview, and coverage details are available throughout the next sections of the report.

The ultimate goal of the audit was to provide RTLabs with a detailed summary of the findings, including any identified vulnerabilities, and recommendations to mitigate any discovered risks. The results of the audit are presented in detail further below.

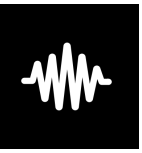


System Overview

Demether is a cutting-edge multichain protocol designed to maximize yield across different blockchain networks. By leveraging a sophisticated blend of restaking, stablecoins, and other financial derivatives, Demether ensures efficient and secure high-yield opportunities for its users. Demether is designed for users seeking to maximize their ETH yields across multiple blockchain layers while minimizing risk and complexity.

The system is composed of several components: deposits managers that handle user deposits in native Ether and WETH, a liquidity pool on the layer-1 chain to handle the staking of user deposits, a protocol token to represent user shares and a messenger contract to handle all bridging interactions.

As a general workflow, a user is able to deposit Ether into the protocol and have the respective shares minted to his address. The shares can be bridged to layer-1 and layer-2 chains and the liquidity can be staked into a liquidity pool using frxEth and EigenLayer.



Repository Coverage and Quality

Code	Tests	Documentation
7 / 10	7 / 10	8 / 10

Resonance's testing team has assessed the Code, Tests, and Documentation coverage and quality of the system and achieved the following results:

- The code follows development some best practices and makes use of known patterns, standard libraries, and language guides. It is easily readable and uses the latest stable version of relevant components. Overall, **code quality is good.**

- Unit and integration tests are included. The tests cover both technical and functional requirements. Code coverage is undetermined. Overall, **tests coverage and quality is good**.
- The documentation includes the specification of the system, technical details for the code and some relevant explanations of workflows and interactions. Overall, **documentation coverage and quality is good**.

Target

The objective of this project is to conduct a comprehensive review and security analysis of the smart contracts that are contained within the specified repository.

The following items are included as targets of the security assessment:

- Repository: `demether-xyz/demether-ethereum/src`
- Hash: 8978ab328f992c61c3df89e5c4d9c0fc23cf3a09

The following items are excluded:

- External and standard libraries
- Files pertaining to the deployment process
- Financial related attacks

Methodology

In the context of security audits, Resonance's primary objective is to portray the workflow of a real-world cyber attack against an entity or organization, and document in a report the findings, vulnerabilities, and techniques used by malicious actors. While several approaches can be taken into consideration during the assessment, Resonance's core value comes from the ability to correlate automated and manual analysis of system components and reach a comprehensive understanding and awareness with the customer on security-related issues.

Resonance implements several and extensive verifications based off industry's standards, such as, identification and exploitation of security vulnerabilities both public and proprietary, static and dynamic testing of relevant workflows, adherence and knowledge of security best practices, assurance of system specifications and requirements, and more. Resonance's approach is therefore consistent, credible and essential, for customers to maintain a low degree of risk exposure.

Ultimately, product owners are able to analyze the audit from the perspective of a malicious actor and distinguish where, how, and why security gaps exist in their assets, and mitigate them in a timely fashion.

Source Code Review - Solidity EVM

During source code reviews for Web3 assets, Resonance includes a specific methodology that better attempts to effectively test the system in check:

1. Review specifications, documentation, and functionalities
2. Assert functionalities work as intended and specified
3. Deploy system in test environment and execute deployment processes and tests
4. Perform automated code review with public and proprietary tools
5. Perform manual code review with several experienced engineers
6. Attempt to discover and exploit security-related findings
7. Examine code quality and adherence to development and security best practices
8. Specify concise recommendations and action items
9. Revise mitigating efforts and validate the security of the system

Additionally and specifically for Solidity EVM audits, the following attack scenarios and tests are recreated by Resonance to guarantee the most thorough coverage of the codebase:

- Reentrancy attacks
- Frontrunning attacks
- Unsafe external calls
- Unsafe third party integrations
- Denial of service
- Access control issues

- Inaccurate business logic implementations
- Incorrect gas usage
- Arithmetic issues
- Unsafe callbacks
- Timestamp dependence
- Mishandled panics, errors and exceptions

Severity Rating

Security findings identified by Resonance are rated based on a Severity Rating which is, in turn, calculated off the **impact** and **likelihood** of a related security incident taking place. This rating provides a way to capture the principal characteristics of a finding in these two categories and produce a score reflecting its severity. The score can then be translated into a qualitative representation to help customers properly assess and prioritize their vulnerability management processes.

The **impact** of a finding can be categorized in the following levels:

1. Weak - Inconsequential or minimal damage or loss
2. Medium - Temporary or partial damage or loss
3. Strong - Significant or unrecoverable damage or loss

The **likelihood** of a finding can be categorized in the following levels:

1. Unlikely - Requires substantial knowledge or effort or uncontrollable conditions
2. Likely - Requires technical knowledge or no special conditions
3. Very Likely - Requires trivial knowledge or effort or no conditions

		Likelihood		
		Very Likely	Likely	Unlikely
Impact	Strong	Critical	High	Medium
	Medium	High	Medium	Low
	Weak	Medium	Low	Info

Repository Coverage and Quality Rating

The assessment of Code, Tests, and Documentation coverage and quality is one of many goals of Resonance to maintain a high-level of accountability and excellence in building the Web3 industry. In Resonance it is believed to be paramount that builders start off with a good supporting base, not only development-wise, but also with the different security aspects in mind. A product, well thought out and built right from the start, is inherently a more secure product, and has the potential to be a game-changer for Web3's new generation of blockchains, smart contracts, and dApps.

Accordingly, Resonance implements the evaluation of the code, the tests, and the documentation on a score **from 1 to 10** (1 being the lowest and 10 being the highest) to assess their quality and coverage. In more detail:

- Code should follow development best practices, including usage of known patterns, standard libraries, and language guides. It should be easily readable throughout its structure, completed with relevant comments, and make use of the latest stable version components, which most of the times are naturally more secure.
- Tests should always be included to assess both technical and functional requirements of the system. Unit testing alone does not provide sufficient knowledge about the correct functioning of the code. Integration tests are often where most security issues are found, and should always be included. Furthermore, the tests should cover the entirety of the codebase, making sure no line of code is left unchecked.
- Documentation should provide sufficient knowledge for the users of the system. It is useful for developers and power-users to understand the technical and specification details behind each section of the code, as well as, regular users who need to discern the different functional workflows to interact with the system.

Findings

During the security audit, several findings were identified to possess a certain degree of security-related weaknesses. These findings, represented by unique IDs, are detailed in this section with relevant information including Severity, Category, Status, Code Section, Description, and Recommendation. Further extensive information may be included in corresponding appendices should it be required.

An overview of all the identified findings is outlined in the table below, where they are sorted by Severity and include a **Remediation Priority** metric asserted by Resonance's Testing Team. This metric characterizes findings as follows:

- **"Quick Win"** Requires little work for a high impact on risk reduction.
- **"Standard Fix"** Requires an average amount of work to fully reduce the risk.
- **"Heavy Project"** Requires extensive work for a low impact on risk reduction.

RES-01	No Threshold Or Heartbeat Update Mechanisms Implemented To Combat Staleness		Resolved
RES-02	Payable Function May Lead To Minting Of Less Tokens		Resolved
RES-03	Unnecessary Usage Of receive() Function		Resolved
RES-04	Possible To Bypass DepositsManager And Call LiquidityPool functions directly		Resolved
RES-05	Missing Setter For Variable _minter		Resolved
RES-06	Insufficient Validation Of _fee		Resolved
RES-07	Sending _fee On Local Minting May Cause Loss Of DETH		Resolved
RES-08	Function _convertToShares Vulnerable To Denial Of Service		Acknowledged
RES-09	Missing Zero Address Validation Of token		Resolved
RES-10	Incorrect Usage Of Function __Ownable_init()		Resolved
RES-11	Missing Zero Address Validation Of sfrxETH		Resolved
RES-12	Missing Zero Address Validation Of messenger		Resolved
RES-13	Missing Validation Of msg.value		Resolved
RES-14	Incorrect Usage Of Initializing Functions		Resolved

RES-15	Incorrect Function Visibility of Initializing Functions	▪▪▮▪▪▪	Resolved
RES-16	Redundant Code On Arithmetic Operations	▪▪▮▪▪▪	Resolved

No Threshold Or Heartbeat Update Mechanisms Implemented To Combat Staleness

Critical **RES-DMTH-PRO01** Business Logic **Resolved**

Code Section

- src/DepositsManagerL1.sol#L145-L157
- src/DepositsManagerL2.sol#L193-L206
- src/DepositsManagerL2.sol#L170

Description

The protocol implements bridging features between layer-1 and layer-2 blockchains to sync the rate at which shares of tokens are minted for the users. By calling the function `syncRate()` on the contract `DepositsManagerL1`, a message is sent to several layer-2 chains to sync their rates with the new rate calculated on the layer-1 chain. On the layer-2 chains, when such a message is received, the rate is updated while the variable `rateSyncBlock` tracks the last block of when the updated occurred.

When minting tokens on a layer-2 chain, the function `getConversionAmount()` is called to calculate how many shares of the token should be minted for the user based on the current rate. The only verification being made simply guarantees that a single rate update was performed in the past. It does not guarantee that the rate is up to date. This opens up the possibility of financial attacks by malicious users that can take advantage of stale rates to take advantages during minting.

Recommendation

It is recommended to implement a proper update mechanism that takes into consideration the staleness of the rate. Such mechanism may be implemented based on rate thresholds for a predefined amount or heartbeat verifications.

Status

The issue has been fixed in 79b1dd095b9c8a2d9a27623e5058622c255bf64b.

Payable Function May Lead To Minting Of Less Tokens

High | RES-DMTH-PRO02 | Business Logic | Resolved

Code Section

- src/DepositsManagerL2.sol#L98

Description

The function `deposit()` is marked as payable, allowing for transfer of native Ether into the smart contract. This function further calls the internal function `_deposit()` while providing `_amountIn` as an input parameter. This effectively means that the amount of minted tokens for the user will solely depend on the provided input parameter `_amountIn`. If the user mistakenly sends Ether when calling this function, they will not be able to mint those funds as tokens.

Recommendation

It is recommended to remove the payable modifier of the function `deposit()`, only allowing the direct deposit of Ether via the function `depositEth()`.

Status

The issue has been fixed in dc99eabaf4d975b722355a4e3e5098db3c99a610.

Unnecessary Usage Of receive() Function

Medium | RES-DMTH-PRO03 | Business Logic | Resolved

Code Section

- src/LiquidityPool.sol#L238

Description

The `LiquidityPool` contract is implementing a `receive` function. This indicates that this contract can successfully receive a native balance transfer. However, the fact that this `receive` implementation is actually empty makes such transfers be out-of-flow. Such transfers can lead to vulnerabilities related to the accounting in the extreme cases. However, they also can cause an inherent loss for any legitimate user who was tricked or made a genuine mistake. Such transfers will not be accounted for accordingly by the contracts so users will lose their native tokens without being able to use the protocol.

Recommendation

It is recommended to remove the unnecessary `receive` function.

Status

The issue has been fixed in 12379fd731038d171228ff7dc6e8ac5d54c5de05.

Possible To Bypass DepositsManager And Call LiquidityPool functions directly

Medium | RES-DMTH-PRO04 | Access Control | Resolved

Code Section

- src/LiquidityPool.sol#L96-106

Description

The Demether protocol defines a specific intended flow for adding liquidity to the protocol. Namely, in order to deposit ETH, users will need to call the `depositETH` function implemented in the `DepositsManagerL1` contract. This function implements appropriate checks, fee calculation and token minting before it calls the `addLiquidity` function defined in the `LiquidityPool` contract. The `addLiquidity` function is mostly responsible for creating shares in the `LiquidityPool` contract. However, the `addLiquidity` function does not implement any access control that would allow it to be executed only if called by `DepositsManagerL1` contract. As a consequence, anyone can call the `addLiquidity` function directly effectively bypassing the accounting done in `DepositsManagerL1`.

Similarly, the `processLiquidity` function defined in the `LiquidityPool` contract can be called directly, without calling the `processLiquidity` function defined in the `DepositsManagerL1` contract.

Recommendation

It is recommended to implement an access control mechanism that will allow only the `DepositsManagerL1` contract to call the `addLiquidity` and `processLiquidity` functions in the `LiquidityPool` contract.

Status

The issue has been fixed in f60e848172e442fc8138b87f59cf75588e322f72.

Missing Setter For Variable _minter

Medium | RES-DMTH-PRO05 | Business Logic | Resolved

Code Section

- src/DOFT.sol

Description

The variable `_minter` does not implement a setter function. If the entity responsible for controlling the address pointed by the variable `_minter` becomes compromised, the entire protocol may be at risk, and may ultimately lead to total loss of funds.

Recommendation

It is recommended to implement a setter function for the variable `_minter` to ensure if it were ever to get compromised, the protocol could update the variable and continue normal operations.

Status

The issue has been fixed in f2f74b3c75ddeed8e581c5aef53ef5a1531f5082.

Insufficient Validation Of _fee

Medium | RES-DMTH-PRO06 | Data Validation | Resolved

Code Section

- src/LiquidityPool.sol#L263-L267
- src/DepositsManagerL2.sol#L210-L214

Description

The fees used within the protocol are not sufficiently validated, and are therefore not restrictive enough of their upper and lower bounds. The fees can both be set to the max value and to the value `0`, both of them being dangerous to the protocol that can eventually cause loss of funds, or confidence from users of the platform.

Recommendation

It is recommended to define a specific range that protocol fees need to be inside of. Then, each relevant function should implement an assertion that the fee value resides in a specified range.

Status

The issue has been fixed in 235f686ff87d951f6d1ef05e5ef76f0ab0d49402.

Sending _fee On Local Minting May Cause Loss Of DETH

Medium | RES-DMTH-PRO07 | Business Logic | Resolved

Code Section

- src/DepositsManagerL1.sol#L75
- src/DepositsManagerL2.sol#L114

Description

The local minting feature of both `DepositsManagerL1` and `DepositsManagerL2` contracts does not require a fee, since the fee should only be used for bridging purposes. However, it is possible to send a non-zero fee via the input parameter `_fee`, effectively allowing for users to lose unnecessary funds.

Recommendation

It is recommended to implement validations to ensure local minting processes do not charge fees.

Status

The issue has been fixed in ebfb2e383734b8909cc860e1f0fc38ba0a220dc0.

Function _convertToShares Vulnerable To Denial Of Service

Low | RES-DMTH-PRO08 | Business Logic | Acknowledged

Code Section

- src/LiquidityPool.sol#L181

Description

The function `_convertToShares()` is used by `addLiquidity()` to calculate the total amount of shares supply on the protocol. The calculation of the supply is proportional to the value tracked by the variable `totalPooledEther`. Not only does this variable increase via `addLiquidity()`, but also via `receive()` and `selfdestruct()`. When any of the latter situation happens, the variable `supply` does not increase. For specially crafted transactions, it may be possible for a malicious user to deny the smart contract from servicing legitimate users.

It should be noted that the impact of this vulnerability directly depends on the amount of Ether the malicious user is willing to spend to grief other users.

Recommendation

It is recommended to implement an accounting mechanism for the shares supply only taking into consideration the amount of Ether that was deposited via `addLiquidity()`. This could be achieved with the use of token smart contracts that track native assets, e.g. `WETH`.

Status

The issue was acknowledged by Demether's team. The development team stated "While the precision issue identified can theoretically be exploited under specific conditions—primarily early in the protocol's lifecycle—the practical impact is significantly limited. The resources required to exploit this vulnerability would be disproportionate to any potential gains. Additionally, an early seeding of the protocol by the project will further mitigate this risk, making the issue implausible in practice. Thus, we accept the resolution with the reduced severity and acknowledge that it may not be worth the heavy refactoring necessary to completely resolve this.".

Missing Zero Address Validation Of token

Low | **RES-DMTH-PRO09** | Data Validation | **Resolved**

Code Section

- src/DepositsManagerL1.sol#L82
- src/DepositsManagerL1.sol#L94
- src/DepositsManagerL1.sol#L111
- src/DepositsManagerL2.sol#L133
- src/DepositsManagerL2.sol#L144
- src/DepositsManagerL2.sol#L161

Description

Throughout the DepositsManagerL1 and DepositsManagerL2 smart contracts the `token` state variable is used to perform external calls. However, this variable is not validated against the Zero Address, allowing for undefined behavior within the protocol.

Recommendation

It is recommended to perform a validation against the Zero Address to ensure external calls are handled properly and successfully.

Status

The issue has been fixed in 9a190fb1fbb6682e87d698bd24b0d02c94bae931.

Incorrect Usage Of Function __Ownable_init()

Low | RES-DMTH-PRO10 | Business Logic | Resolved

Code Section

- src/LiquidityPool.sol#L83

Description

The function `initialize()` on the upgradeable smart contract `LiquidityPool` does not call its parent's `OwnableAccessControl` initializer function `__OwnableAccessControl_init()`. Instead, it is calling `__Ownable_init()`.

While it is possible to call some parents' initializer functions out of order or higher in the inheritance chain, it is not recommended to do so as it will lead to development errors and undefined behavior within the protocol.

Recommendation

It is recommended to only call immediate parents' initializer functions in the proper inheritance order and let them call their parents' initializer functions.

Status

The issue has been fixed in 07c47c69be099ca5fa08f02546bbd39ee02daa4f.

Missing Zero Address Validation Of sfrxETH

Low | **RES-DMTH-PRO11** | Data Validation | **Resolved**

Code Section

- src/LiquidityPool.sol#L147

Description

Throughout the LiquidityPool smart contract the `sfrxETH` state variable is used to perform external calls. However, this variable is not validated against the Zero Address, allowing for undefined behavior within the protocol.

Recommendation

It is recommended to perform a validation against the Zero Address to ensure external calls are handled properly and successfully.

Status

The issue has been fixed in ccc1fd0255ccf8f2d9b6d1e5803b87964d856870.

Missing Zero Address Validation Of messenger

Low | RES-DMTH-PRO12 | Data Validation | Resolved

Code Section

- src/DepositsManagerL2.sol#L136
- src/DepositsManagerL2.sol#L187

Description

Throughout the DepositsManagerL2 smart contract the `messenger` state variable is used to perform external calls. However, this variable is not validated against the Zero Address, allowing for undefined behavior within the protocol.

Recommendation

It is recommended to perform a validation against the Zero Address to ensure external calls are handled properly and successfully.

Status

The issue has been fixed in f4049014f1cf1e6e3c1de3accd2964aa4bbaf696.

Missing Validation Of msg.value

Low | **RES-DMTH-PRO13** | Data Validation | **Resolved**

Code Section

- src/DepositsManagerL2.sol#L114

Description

The function `depositETH()` does not validate the Ether received from the user identified within the global variable `msg.value`. This variable is then used in an arithmetic operation along with another input variable `_fee`, which is controlled by the user. This effectively means that it may be possible for a user to trigger an integer underflow condition.

Recommendation

It is recommended to validate both the input parameter `_fee` and the Ether sent by the user, and only allow arithmetic operations between filtered and intended values.

Status

The issue has been fixed in 864dbae8c33bfb8180416ebb3a85ee29e9d1350d.

Incorrect Usage Of Initializing Functions

Info | **RES-DMTH-PRO14** | Code Quality | **Resolved**

Code Section

- src/OwnableAccessControl.sol#L48-L52
- src/DepositsManagerL1.sol#L54-L60
- src/DepositsManagerL2.sol#L75-L85

Description

The functions `_init()` and `_init_unchained()` are an implementation of OpenZeppelin's Upgradeable Proxies design standard and are used to serve as the constructor of upgradeable contracts. Despite being both used as initializers, they possess slightly different use cases to correctly implement the analog linearization of smart contract constructors.

The function `_init()` should be used and implemented to embed the linearized calls to all parent initializers. The function `_init_unchained()` should be used to perform the initialization of the variables for the current contract only.

As a consequence of this design, it is possible that, due to the lack of automatic linearization, the calling of `_init()` functions initializes the same contract multiple times. For these cases, the `_init_unchained()` function may be used manually to avoid double initialization, however it will break the upgradeability design and compatibility between smart contracts on the blockchain.

Additionally, the inherited smart contracts should be initialized according to their inheritance order, from the most base-like to the most derived.

There are multiple instances where contracts do not follow the design standard in the following account:

- `_init_unchained()` function is not used.
- Initialization according to inheritance order is incorrect

Recommendation

It is recommended to follow OpenZeppelin's Upgradeable Proxies design standard to the best of the possibilities, and follow the necessary recommendations to maintain composability, interoperability, and consistency across the blockchain.

It should be noted however, that it is not always possible to follow the recommendations to their full extend due to double initialization issues, and in those cases it may be required to manually initialize the parent contracts.

Status

The issue has been fixed in 8ec3e47376fa618386154e64ea90d1857ea8a73e.

Incorrect Function Visibility of Initializing Functions

Info | **RES-DMTH-PRO15** | Code Quality | **Resolved**

Code Section

- src/OwnableAccessControl.sol#L48-L52

Description

Smart contract upgradeability using OpenZeppelin's standard, requires contracts to use initializer functions. The function to use on child contract should be named `initialize()` and should possess the `initializer` modifier, while parent contracts should have `_init()` and `_init_unchained()` functions with the modifier `onlyInitializing`, where there would otherwise be constructors called automatically on non-upgradeable smart contracts.

The initializer functions on the parent smart contracts are meant to be initialized by the child contract, as such, their visibility can be constrained to `internal`. The following functions do not implement the proper visibility to encourage a principle of least privilege:

- The visibility of the function `__OwnableAccessControl_init()` is incorrectly set as `public`.

Recommendation

It is recommended to assign a function's visibility according to the necessary privileges required to call that function. The visibility of parent's initializer function on inherited upgradeable smart contracts should be set to `internal`.

Status

The issue has been fixed in 8ec3e47376fa618386154e64ea90d1857ea8a73e.

Redundant Code On Arithmetic Operations

Info	RES-DMTH-PRO16	Code Quality	Resolved

Code Section

Not specified.

Description

Throughout the source code of the protocol, there are several instances where `uint256` variables are being compared to `0`. The comparison of these variables being lesser than or equal to `0` is redundant as `uint256` variables can only contain `0` or positive values.

Recommendation

It is recommended to remove the redundant checks to improve code readability and gas usage.

Status

The issue has been fixed in 9de3c2a1dafe9e3811a51108d76ea7bf1d22c641.

Proof of Concepts

RES-08 Function _convertToShares Vulnerable To Denial Of Service

LiquidityPool.t.sol (added lines):

```
...

contract AddLiquidityTest is TestSetup {
    ...

    function test_exploit1() external {
        vm.deal(address(depositsManagerL1), 1 ether);
        vm.prank(address(depositsManagerL1)); // Using the authorized caller
        liquidityPool.addLiquidity{value: 100 wei}();
        assertEq(liquidityPool.totalShares(), 100 wei);

        // Malicious user adds ETH to liquidityPool
        vm.deal(address(liquidityPool), 12_000 wei);

        // Next deposit will revert
        vm.prank(address(depositsManagerL1)); // Using the authorized caller
        liquidityPool.addLiquidity{value: 100 wei}();
    }
}

...
```